PSpec-SQL: Enabling Fine-Grained Control for Distributed Data Analytics

Chen Luo, Fei He, Fei Peng, Dong Yan, Dan Zhang, Xin Zhou

Abstract—Business organizations regularly collect customer data to improve their services. Organizations may want to share data within themselves or even with third-parties to maximize data utility. Since business data contain lots of customer data, organizations must respect customers' privacy expounded by privacy laws. In this paper, we present PSpec-SQL, a distributed data analytics system that automatically enforces privacy compliance for SQL queries. Our system provides a high-level language PSpec for the data owner to specify her data usage policy. As usual, the data analyst queries data to perform data analysis, but our system checks each query to ensure only policy-compliant queries are executed.
We have implemented a prototype of PSpec-SQL on top of Spark-SQL, and carried out a case study on the TPC benchmarks. The results show the practicability of our system with negligible overhead over query processing.

Index Terms—Access Control, Data Analytics, Privacy Protection.

1 Introduction

Recent advances in cloud computing bring great opportunities for many small and medium enterprises to discover values from the business data with data analytics techniques [1]. Cloud providers like Amazon Web Service-have already provided data analytics platforms as cloud services. Instead of building IT infrastructures on their own, the enterprises can directly leverage cloud infrastructures and outsource data analytics tasks to third party data analysts to perform data analysis.

However, when analyzing business data, one major concern of the enterprises is the privacy issue. As the business data contains lots of private information such as identities and personal preferences, misusing such data can cause privacy breaches and severely degrade the enterprise's reputation. A traditional technique to protect customer's privacy [2] is based on syntactic anonymization. Through anonymization, data related to private information are removed or generalized. But anonymization lacks flexibility since the data is anonymized beforehand and fails to adjust the data privacy and utility based on the actual needs of different data analytics tasks.

The privacy issue is even more intractable with the introduction of distributed computing into the working environment. From the data providers' viewpoint, their private data are computed on remote, unknown, and potentially untrusted machines. From the data analysts' viewpoint, the input data may originate from multiple sources. Traditional techniques for centralized systems are thus not sufficient. Privacy protection for distributed data analytics is demanded.

- *Fei He (corresponding author) is with School of Software, Tsinghua University, and also with KLiss, MoE.*
- *Chen Luo and Fei Peng were with School of Software, Tsinghua University when they participated in this work.*
- *Dong Yan, Dan Zhang and Xin Zhou were with Intel Labs China when they participated in this work.*

Manuscript received XX XX, 201X; revised XX XX, 201X.

There exist some distributed big data analytics systems, e.g., Spark-SQL [3] and Hive [4]. These systems greatly hide the complexities of distributed systems, while providing simple SQL-like data models and query languages for efficient large-scale data analytics. Most existing distributed data analytics systems share a similar workflow of query execution. The submitted query is first parsed and analyzed in the master node, and then distributed to a set of worker nodes. These worker nodes concurrently execute to produce the final results.

Differential privacy [5], [6] is a privacy protection technique adopted in some distributed data analytics systems. A carefully designed curator resolves a pre-determined set of queries from third parties. Through noisy answers, one can argue that private information cannot be leaked through the curator. Designing differentially private curators however requires tedious analysis and mathematical insights. Adopting differential privacy in organizations with varying business data can be too challenging at present.

Thus, a more practical approach is to let the data owner specify data use restrictions to avoid the misuse of privacy-related data. Similar to differential privacy, a curator is used to mediate data and queries. If a query does not access the data according to these restrictions, the curator immediately rejects the query. One central requirement for the specification language is to get a good balance between data privacy and data utility, such that the customer's privacy is well protected while the data is still usable.

Conventional access control languages, such as EPAL [7] and XACML [8], are not suitable for privacy protection in data analytics. A data usage specification may involve complicated trigger conditions and intricate handling, which can readily exceed the expressiveness of these languages. Consider the sales information of an online retail company. The company may be willing to share the amounts of all transactions last year. It may be hesitant to share the amounts of all transactions associated with an IP address last year. Instead of blindly rejecting access of sensitive information, the company may release such sensitive information after *desensitization*. For instance, transaction amounts associated with *truncated* IP addresses can be shared, as well as *average* transaction amounts associated with full IP addresses.

Purpose-based access control techniques [9], [10], [11], [12], [13], which extend the traditional database systems to enforce privacy policies, are still not fully applicable to the distributed data analytics. These techniques allow the data owner to declaratively specify data access privileges in the database systems. However, their specifications directly operate on relational data, and need be written by database experts. To adhere to privacy laws and policies, privacy specifications are more expected to be handled by legal experts [14], who have little background in databases, and the specification language is thus necessarily at an abstract level to hide the details of data models. Such abstraction also benefits organizations by allowing specification reuse as personal-related data concepts are often shared in spite of various data sources. Furthermore, the semi-structured complex data types, e.g., struct, array and map, supported by distributed data analytics systems further complicate the policy enforcement process.

Moreover, all previous works in access control techniques [7], [8], [11], [12], [13] consider information being independent of each other, i.e., accessing multiple pieces of information together is equivalent to accessing each piece separately. However, this is not the case for data analytics because of the association among information. For example, access customer's address and health condition separately may be acceptable, while access them together will cause severe privacy breach since it directly leaks health condition for each customer.

The data usage specification alone does not guarantee compliance unless it can be automatically enforced. As the first step towards enforcement, the gap between the abstract concepts in the specification and the underlying data (relational data in this paper) should be filled through a *data labeling* process. For the performance consideration, we consider column-level labeling so that queries can be statically checked with little performance overhead. This is important for today's distributed big data systems, which are often targeting at large-scale and long running data analytics queries. Nevertheless, data labeling is non-trivial for two issues. Sometimes the proper meanings of some columns cannot be determined statically, but rather depend on how the table is accessed with other tables. For example, the *Address* table may contain both customer addresses and store addresses, and such ambiguity can only be resolved when the query is present. Moreover, the complex data types as mentioned above further complicate the data labeling process. But once the data is properly labeled, the privacy specification can be enforced against submitted queries with program analysis techniques.

Putting things together, we present PSpec-SQL, a distributed data analytics system which automatically enforces data-usage compliance. PSpec-SQL is built upon Spark-SQL[1], and provides a simple and high-level specification language PSpec for the data owner to specify data usage restrictions. At runtime, each submitted query is checked to ensure only specification-compliant queries are executed.

Contributions. Our main contributions in this paper are summarized as follows:

- We introduce PSpec, a high-level data usage specification language for distributed data analytics systems, with explicit support for data association and desensitization.
- To link the PSpec concepts with the underlying data, we provide comprehensive support for labeling relational data.
- We present the checking algorithm for SQL queries with information flow analysis techniques.
- Finally, to evaluate our system, we carried out a case study on three industry standard benchmarks [15]. The results show the usability and practicability of our system with negligible overhead over query execution.

A preliminary discussion of the PSpec language was included in a conference paper [16] by the same authors. Compared to [16], this paper makes the following new contributions. Firstly, [16] was focused on the PSpec language only, while this paper presents a PSpec-based distributed data analytics system that automatically enforces privacy compliance against SQL queries. Among the four contributions summarized in the above paragraph, the second and third ones are completely new compared to [16]. Secondly, even for the PSpec language, it is only sketched at a very high level in [16]. In contrast, this paper gives a full description of PSpec, including its formal syntax and semantics.

The rest of the paper is organized as follows. Section 2 presents an overview of our system. Sections 3, 4, and 5 introduce the details of the PSpec language, data labeling, and the privacy checking algorithm respectively. Section 6 reports our implementation and evaluation. Section 7 further discusses some possible extensions of our system and Section 8 briefly reviews the related works. Finally, Section 9 concludes this paper.

2 System Overview

In this section, we discuss a motivating example and present an overview of our system.

2.1 Motivating Example

As a motivating and running example, consider a retail company that owns customer and sales data. The data schema is shown in Fig. 1, where the primary keys are noted with parentheses and the foreign key references are represented by arrows. Each table is briefly explained as follows.

- *Customer*: stores customers' personal information, and customer addresses are stored in the *Address* table.
- *Store_Sales*: stores sales information, and refers to the *Customer*, *Store*, and *Item* tables.
- *Item*: stores items information.
- *Store*: contains stores information, and store addresses are stored in the *Address* table.

1. https://spark.apache.org/sql/

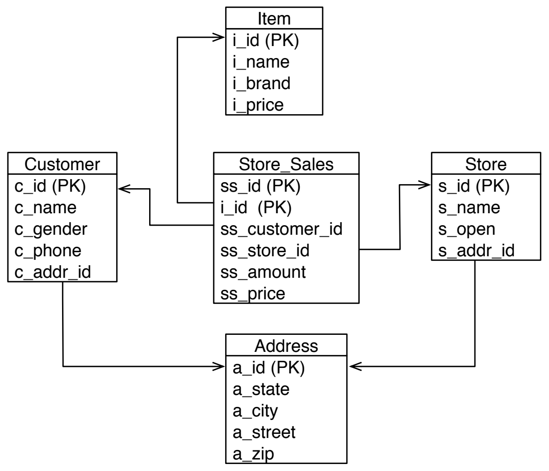


Fig. 1. Example Schema

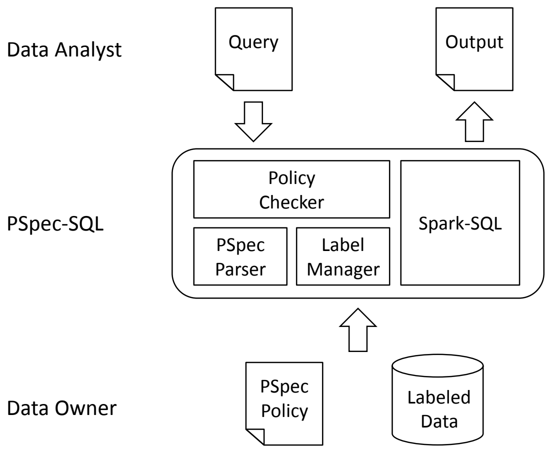


Fig. 2. System Overview

- *Address*: stores the addresses for customers and stores.

With the business data at hand, the company lets data analysts to query the data to perform data analysis. However, one major concern of the retail company is that the customer data (the *Customer* and *Address* tables) may be improperly used by the data analyst. For example, the malicious data analyst may query all customer names with living addresses and sell them to other companies, which directly violates related privacy laws and causes severe privacy breaches. Thus, to ensure the data are properly used, the legal team in the company enacts some privacy-related data use policies, which are expected to be automatically enforced by data analytics systems.

2.2 PSpec-SQL Overview

The scenario considered in this paper mainly involves the following logical participants:

- The data owner, e.g., the retail company, who owns the business data.
- The data analyst, who queries the business data to perform data analysis.
- The data analytics system, which manages the business data and allows the analyst to submit queries.

Fig. 2 depicts an architecture overview of our data analytics system, i.e. PSpec-SQL. PSpec-SQL is built upon Spark-SQL, and contains several new modules, including the PSpec parser, label manager and policy checker. The PSpec parser parses the PSpec policy defined by the data owner, and the label manager manages a set of labels which connect the abstract data concepts in PSpec with the relational data. With the PSpec policy and the labeled data set, the policy checker checks each submitted query during query processing phase to ensure the specification compliance of submitted queries.

We further elaborate the work flow of our system with the motivating example. As mentioned before, to ensure the data are properly used, the legal team of the retail company may have enacted several data use policies according to the related policy and laws. Typical policies include "the direct output of the *customer name* is forbidden", "only the aggregated *sales price* is allowed when outputted with *personal information* together" and "the use of the customer living *state*, *city*, and *street* together is forbidden". The data owner may also adjust the data use requirements based on the needs of the data analysis tasks. With our PSpec language, these policies can be easily encoded into the enforceable PSpec policies, as shown in Section 3

Since PSpec only contains abstract data concepts, such as the *customer name* and *sales price*, the company then needs to label the relational data in Fig. 1 with the corresponding data concepts. For example, the *name* column in the *Customer* table is labeled with the *customer name*, while the *ss_price* column in the *Store_Sales* table is labeled with the *sales price*. Note that data labeling only happens at schema level, which will not incur too much human effort.

Finally, with the PSpec policy and the labeled data set, the policy checker performs information flow analysis on submitted queries to detect possible violations. For example, the following query is stopped since it outputs the *state*, *city*, and *street* together, which directly violates the third rule above.

```
SELECT state, city, street, avg(ss_price)
FROM Customer JOIN Store_Sales ON c_id = ss_customer_id
              JOIN Address ON c_addr_id = a_id
GROUP BY state, city, street
```

Note that our system requires no extra effort from the data analyst. To perform data analysis, the data analyst only needs to write queries conforming to the data use policies as usual.

3 Specification Language

In this section, we present our specification language PSpec. As mentioned, since PSpec targets at legal experts, who may have little background in databases, it is thus designed as a high-level language based on abstract concepts. Moreover, PSpec is designed for specifying data use restrictions in the distributed data analytics systems. Most of these systems provide a SQL-like query language, which allows the user to apply a sequence of transformations to produce the final results. The PSpec rules thus specify restrictions on these data accesses and transformations in order for privacy protection.

In the remainder of this section, we first present the language syntax as well as the design choices, then introduce the formal semantics.

3.1 PSpec Syntax

Since PSpec is based on abstract concepts, we separate PSpec into the *vocabulary* part and *policy* part as in EPAL [7]. Briefly, the vocabulary part defines abstract concepts while the policy part defines a set of the PSpec rules.

3.1.1 Vocabulary

In the vocabulary part, the data owner defines a set of user categories and data categories. A user category represents a role of the analyst. A data category represents a privacy-related data concept. Both user and data categories are hierarchical. In a category hierarchy, a parent category is the generalization of its child categories; a child category is a specialization of its parent category. When a parent category is referred, all its descendants are also referred.

Fig. 3 shows category hierarchies for the retail company. The top-level user category is *Analyst*, which represents all data analysts. *Analyst* is divided into *Report Analyst*, *Marketing Analyst*, and *Advertise Analyst* based on the tasks assigned to them. The data categories are classified as *Key Attribute*, *Quasi Identifier*, and *Sensitive Attribute*. *Key Attribute* denotes the attributes that can uniquely identify an individual, such as *Name* and *Phone* etc. *Quasi Identifier* represents the attributes that may locate individuals by association, such as the combination of *Birth*, *Zip*, and *Gender* [2]. Finally, *Sensitive Attribute* carries sensitive personal information, such as customer sales records.

The data owner should also identify and specify all supported desensitization operations for data categories. Informally, desensitize operations can be used to make sensitive data categories ready for access. For example, the desensitize operation for *Sale_Price* can be an aggregate operation, such as *Min*, *Max*, and *Avg* etc., while for *Zip* it can be the *truncate* operation. Techniques from data anonymization [2] and differential privacy [5], [6] can also be employed to realize the desensitization operation. A data category automatically inherits its ancestors' desensitize operations.

3.1.2 Policy

Before introducing details, we discuss data access issues pertaining to data analytics. In data analytics, accessing multiple pieces of information together may not be equivalent to accessing each piece separately because of *association*. For example, accessing a customer's address and salary separately may be acceptable. Yet accessing them together causes a severe privacy breach. Instead of forbidding accessing any sensitive information, *desensitization* is useful to balance between privacy and utility in data analytics. For instance, full IP addresses may locate individuals, and hence are considered harmful; *truncated* IP addresses may be acceptable and useful to data analysis.

The data owner defines a set of PSpec rules to regulate data access. Informally, each rule states under what restrictions can a user category access certain data categories together. In order to describe intricate data usage specifications, PSpec provides explicit support for data association and desensitization. We require a user category to satisfy all the applicable rules. The reason is that usually only a part of the business data is related to customers' privacy. This allows the data owner to focus on privacy-related data.

The rule grammar of PSpec is shown in Fig. 4. Each PSpec rule contains a *scope* part and a *restriction* part (separated by =>). The scope part is defined by `User-Ref` and `Data-Assoc` (short for `Data Association`). `User-Ref` refers a user category (`<user>`) to specify applicable user categories. When a category is referred, all its descendants are also referred except those excluded explicitly. `Data-Assoc` specifies applicable association of data categories by a sequence of `Data-Ref`s. Each `Data-Ref` refers a data category (`<data>`) with an *action*. An action specifies how a data category is accessed. Two types of actions are distinguished in our system since they have different privacy implications. The `projection` action means that the data category is output by a query, and the `condition` action denotes that the data category is used in the control flow of a query. The actions also form a hierarchy where `access` is the parent of `projection` and `condition` (see Fig.5). Note that from the traditional information flow point of view, even the condition action may cause information leakage. We intentionally separate `projection` from `condition` since the former has a stronger implication and let the user to balance the utility and privacy based on their needs. For `Data-Assoc`, data categories referred in each `Data-Ref` must be disjoint, i.e., no referred data category overlaps with each other.

A rule can *forbid* the query or specify desensitize requirements by restrictions. A restriction is a sequence of `Desensitize`s with the same length as the data association of the rule. Each `Desensitize` specifies a set of admissible desensitize operations (`<operation>`), and requires the corresponding data category in the data association be desensitized with one of them ($\emptyset$ denotes no desensitization required). The corresponding data category must support the specified desensitize operations as defined in the vocabulary. A query must satisfy one of these restrictions in an applicable rule. Thus, the restriction part is essentially a disjunction of restrictions, while each restriction is a conjunction of restricted data categories, which makes PSpec expressive enough to specify complex policies.

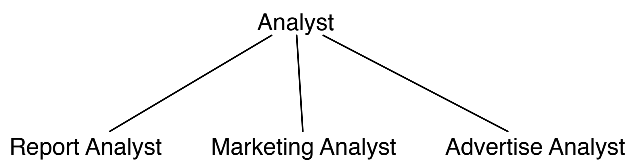
Consider the example rules in Section 2.2 again. For simplicity, we assume all rules are applicable to *Analyst*. First, the company forbids the analyst access any customer's name, which can be formalized as follows:

r_1: `Analyst,[access Name]=>forbid`

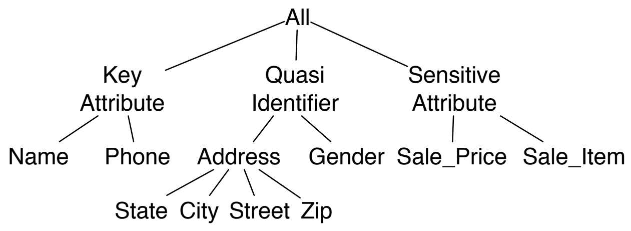
Second, the company requires the sale price be aggregated when output with personal information together.

r_2: `Analyst,[projection All exclude Sensitive Attribute,`
`projection Sale_Price] =>[{},{avg,max,min,sum}]`

In r_2, we define a data association of length two. The first data access refers to the projection on all but sensitive data categories. The second data access refers to the projection on sales price. Rule r_2 further requires *Sale_Price* be aggregated by one of the *avg*, *max*, *min*, or *sum* operations. Note that r_2 refers to data association and cannot be specified by EPAL [7] and XACML [8].



(a) Example User Hierarchy



(b) Example Data Hierarchy

Fig. 3. Example Category Hierarchies

```
Rule       ::= User-Ref, Data-Assoc =>
               ('forbid' | Restriction$_1$,..., Restriction$_m$)
User-Ref   ::= <user> ('exclude' <user>+)?
Data-Assoc ::= '[' Data-Ref$_1$,···,Data-Ref$_n$ ']'
Data-Ref   ::= Action <data> ('exclude' <data>+)?
Action     ::= 'access' | 'projection' | 'condition'
Restriction::= '[' Desensitize$_1$,···,Desensitize$_n$ ']'
Desensitize::= '{'(<operation>$_1$,···,<operation>$_k$)?'}'
```

Fig. 4. PSpec Rule Grammar

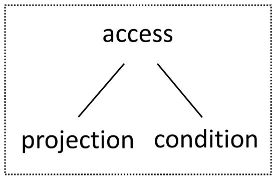


Fig. 5. Action Hierarchy

Finally, the company forbids the access of state, city, and street together.

```
r3: Analyst,[access State,access City,access Street]=>forbid
```

Note that r_3 only forbids access to them together. One can access any one or two of them freely. Again, r_3 cannot be specified by the existing access control languages [7], [8].

The reader may have already noticed that PSpec does not model *access purpose* directly, since it is difficult for data analytics systems to automatically verify the claimed purpose. For simplicity, we assume each data analyst is assigned a proper user category by the system administrator based on her tasks and purposes. However, it is straightforward to extend PSpec with other language elements like *purpose* and *obligation*. This issue is further discussed in Section 7. Also note that a data category is considered desensitized as long as one of the specified desensitize operations is performed. We do not support a sequence of desensitize operations since it complicates both the data owner for writing the PSpec rules and the data analyst for writing proper queries. For this case, the data owner can simply define a new operation that combines all necessary operations inside.

3.2 Formal Semantics

After introducing the syntax of PSpec, now we discuss its formal semantics. Recall that PSpec specifies restrictions on data access, thus the semantics are defined against queries. In the following, we first introduce some notions, and then discuss the semantics of the query and the PSpec policy respectively. All notations are summarized in Table 1.

TABLE 1
Summary of Notations

Notation	Description
$\mathbb{U}_{user}$	the universal set of user categories
$\mathbb{U}_{data}$	the universal set of data categories
$\mathbb{U}_{act}$	the universal set of actions
u	a user category
d	a data category
a	an action
$\langle d, a\rangle$	a data access
$\mathbb{U}_{da}$	the universal set of data accesses
$\overline{\tau}$	an associated data access
$\mathbb{U}^n_{da}$	the set of all associated data accesses
$\mathbb{U}_{op}$	the universal set of desensitize operations
δ	a desensitize operation
$\delta_\emptyset$	the *empty* desensitize operation
Δ	a set of desensitize operations
q	a query
$(\langle d, a\rangle, \delta)$	a data leakage pair
$\lvert q\rvert$	the number of data leakages in q
r	a PSpec rule
$\overline{\xi}$	a rule restriction

Let $\mathbb{U}_{user}$, $\mathbb{U}_{data}$ and $\mathbb{U}_{act}$ be the universal set of all user categories, data categories and actions, respectively. We use $u \in \mathbb{U}_{user}$, $d \in \mathbb{U}_{data}$, and $a \in \mathbb{U}_{act}$ to denote a user category, data category, and action, respectively. Especially, we denote the pair $\langle d, a\rangle$ a *data access*. Let $\mathbb{U}_{da} = \mathbb{U}_{data} \times \mathbb{U}_{act}$ be the set of all data accesses with respect to $\mathbb{U}_{data}$ and $\mathbb{U}_{act}$. An associated data access $\overline{\tau}$ of size n is an n-tuple $\overline{\tau} = (\tau_1, \tau_2, \ldots, \tau_n)$, where $\tau_i \in \mathbb{U}_{da}$ for $1 \leq i \leq n$. Denote $\mathbb{U}^n_{da}$ the set of all associated data accesses of length n. Given two data access tuples $\overline{\tau}$ and $\overline{\tau}'$, we say $\overline{\tau}$ is *included* in $\overline{\tau}'$ (written $\overline{\tau} \preceq \overline{\tau}'$) if all data accesses contained in $\overline{\tau}$ are also contained in $\overline{\tau}'$. Informally, if we treat both $\overline{\tau}$ and $\overline{\tau}'$ as a set, $\overline{\tau} \preceq \overline{\tau}'$ just means $\overline{\tau}$ is a subset of $\overline{\tau}'$.

Let $\mathbb{U}_{op}$ be the universal set of all available desensitize operations. We use $\Delta \subseteq \mathbb{U}_{op}$ to denote a set of desensitize operations, and $\delta \in \mathbb{U}_{op}$ a desensitize operation. Specifically, denote $\delta_\emptyset \in \mathbb{U}_{op}$ the *empty* desensitize operation, representing the corresponding data category can be directly accessed.

From the information flow point of view [17], [18], the semantics of a query q is defined upon a set of data leakages. A *data leakage* is a pair $(\langle d, a\rangle, \delta)$, where $\langle d, a\rangle$ is a data access and δ a desensitize operation. A data leakage $(\langle d, a\rangle, \delta)$ means that the data d is leaked through action channel a after the desensitize operation δ. Let $|q|$ be the number of data leakages in q. For ease of discussion, we extract the data accesses and desensitize operations from the data leakages of q, each as a $|q|$-tuple.

Definition 3.1 (Query)**.** A query q is a triple $q = (q.user, q.asso, q.des)$, where

- $q.user \in \mathbb{U}_{user}$ is the user who submits q,
- $q.asso \in \mathbb{U}_{da}^{|q|}$ is a tuple of data accesses, and
- $q.des = (\delta_1, \delta_2, \ldots, \delta_{|q|})$ is the desensitize operations in q.

To avoid semantic inconsistency, we assume any data category occurring in a query is a leaf node in the data hierarchy. The reason is that a data category is restricted often implies its ancestors are also restricted. For example, *Zip* should be truncated implies *Address* should also be truncated. However, this could lead to semantic inconsistency since the *truncate* operation may not be supported by *Address*. To handle this, we require a query only access leaf data categories, but this will not be a limitation in practice since one can simply label one piece of data with multiple data categories (see Section 4).

For example, consider *Report Analyst* submits a query q_1, which projects *State*, *City*, and aggregated *Sale_Price*. Obviously, $|q_1| = 3$, and

$$\begin{aligned} q_1.user &= Report\ Analyst, \\ q_1.asso &= (\langle State, projection\rangle, \langle City, projection\rangle, \\ &\quad \langle Sale\_Price, projection\rangle), \\ q_1.des &= (\delta_\emptyset, \delta_\emptyset, avg). \end{aligned}$$

Note that $\delta_\emptyset$ is used if one wants to access data without any desensitization.

Since a PSpec policy contains a set of PSpec rules, we first introduce the semantics of a PSpec rule. Note that the associated data accesses referred by a rule r have the same size, denoted as $|r|$. Given a rule r, we use $r.User \subseteq \mathbb{U}_{user}$ and $r.Asso \subseteq \mathbb{U}_{da}^{|r|}$ to denote the set of user categories and the set of associated data accesses applicable to r, respectively. In the grammar of Fig. 4, $r.User$ is specified by `User-Ref`. It corresponds to the descendants of the specified user category minus the descendants of any of the excluded user categories. $r.Asso$ is specified by `Data-Assoc` in the grammar. Let $r.Asso_i$ be the set of data accesses specified by the i-th `Data-Ref` of `Data-Assoc`, which is obtained similarly as $r.User$, $r.Asso$ is the Cartisian product of $r.Asso_i$ for $1 \le i \le n$.

Given an associated data access $\overline{\tau} = (\langle d_1, a_1\rangle, \langle d_2, a_2\rangle, \ldots, \langle d_n, a_n\rangle)$ of size n, a restriction $\overline{\xi}$ is an n-tuple $(\Delta_1, \Delta_2, \ldots, \Delta_n)$, where $\Delta_i \subseteq \mathbb{U}_{op}$ specifies the set of admissible desensitize operations. An operation in Δ_i must be taken to protect d_i against a_i. Given a rule r, we use $r.Res$ to denote the set of restrictions specified in r. Each restriction $\overline{\xi}_i \in r.Res$ is specified by `Restriction`$_i$. in the grammar.

Definition 3.2 (Rule)**.** A rule r is a triple $r = (r.User, r.Asso, r.Res)$, where

- $r.User \subseteq \mathbb{U}_{user}$ is the set of user categories,
- $r.Asso \subseteq \mathbb{U}_{da}^{|r|}$ is the set of associated data accesses, and
- $r.Res = \{\overline{\xi}_1, \overline{\xi}_2, \ldots, \overline{\xi}_m\}$ is the set of restrictions. Especially, $r.Res = \emptyset$ if r is a forbidden rule.

For example, consider the following rule:

```
r4: Report Analyst, [projection Address, projection
     Sale_Price] => [{},{sum,avg,min,max}]
```

Obviously, $|r_4| = 2$, and

$$\begin{aligned} r_4.User &= \{Report\ Analyst\} \\ r_4.Asso &= \{(\langle Address, projection\rangle, \langle Sale\_Price, projection\rangle), \\ &\quad (\langle State, projection\rangle, \langle Sale\_Price, projection\rangle), \\ &\quad (\langle City, projection\rangle, \langle Sale\_Price, projection\rangle), \\ &\quad (\langle Street, projection\rangle, \langle Sale\_Price, projection\rangle), \\ &\quad (\langle Zip, projection\rangle, \langle Sale\_Price, projection\rangle)\} \\ r_4.Res &= \{(\mathbb{U}_{op}, \{sum, avg, min, max\})\} \end{aligned}$$

Note that for ease of discussion, $\mathbb{U}_{op}$ is used if there is no desensitize requirement, i.e., `Restriction`$_i$ in the grammar contains no desensitize operation.

Definition 3.3. A rule r is *applicable* to a query q (or equivalently, *q triggers r*) if

- $q.user \in r.User$, and
- there exists at least one associated data access $\overline{\tau}$ in $r.Asso$, such that $\overline{\tau} \preceq q.asso$.

The first condition in above definition performs user scope check, and the second condition performs the scope check of associated data access. The rule is applicable to the query only if both checks are passed. Moreover, if there exists an $\overline{\tau}$ in $r.Asso$ such that the second condition holds, we further define $q.des|_{\overline{\tau}}$ as the projection of $q.des$ to $\overline{\tau}$, obtained by removing $q.des[i]$ (the i-th element of $q.des$) if $q.asso[i]$ (the i-th element of $q.asso$) does not occur in $\overline{\tau}$ for $1 \le i \le |q|$. The size of $q.des|_{\overline{\tau}}$ equals to $|r|$. Note that there may be more than one $\overline{\tau}$ satisfying the second condition. With different $\overline{\tau}$, the set $q.des|_{\overline{\tau}}$ may be different. We thus define

$$q.des|_r = \{q.des|_{\overline{\tau}} \mid \overline{\tau} \in r.Asso \text{ and } \overline{\tau} \preceq q.asso\}$$

Definition 3.4. A n-tuple of desensitize operations $\overline{\delta} = (\delta_1, \delta_2, \ldots, \delta_n)$ *satisfies* a restriction $\overline{\xi} = (\Delta_1, \Delta_2, \ldots, \Delta_n)$ if $\delta_i \in \Delta_i$ for $1 \le i \le n$.

Definition 3.5 (Rule Satisfaction)**.** A query q *satisfies* a rule r iff either r is not applicable to q or for any $\overline{\delta} \in q.des|_r$, there exists at least one restriction $\overline{\xi} \in r.Res$, such that $\overline{\delta}$ *satisfies* $\overline{\xi}$.

Intuitively, a query satisfies a triggered rule if all data categories accessed by the query are properly desensitized. Consider the rule r_4 and the query q_1 in this subsection. r_4 is applicable to q_1 since $q_1.user \in r_4.User$ and two former associated data accesses in $r_4.Asso$ are included in $q_1.asso$. Hence, $q_1.des|_{r_4} = \{(\delta_\emptyset, \text{avg})\}$. Since $(\delta_\emptyset, \text{avg})$ satisfies the restriction in $r_4.Res$, r_4 is satisfied by q_1 (Definition 3.5).

Definition 3.6 (Policy Satisfaction)**.** A query q *satisfies* a PSpec policy iff q *satisfies* all rules in the policy.

4 Data Labeling

After introducing the specification language PSpec, we shall then discuss how to enforce the PSpec policy automatically. Since the PSpec rules are defined upon abstract data categories, the data owner should first label her data with proper data categories to enable policy enforcement. Currently, we support data labeling on column level so that queries can be statically checked with little performance overhead. Thus, the data owner should label data columns with proper data categories based on column semantics. Labeling one column with multiple data categories is also supported if the column corresponds to multiple data categories. However, recall from Section 3.2, columns can only be labeled with leaf data categories.

Besides labeling columns, the data owner should also provide implementations of the desensitize operations specified in the vocabulary as user-defined functions (UDFs) and register them into our system. The data owner may optionally provide multiple implementations for the same desensitize operation for different columns. For example, consider the *truncate* operation for the *IP* category. Suppose *IP* is stored as dot separated strings in column *IP1*, while integer in column *IP2*. Thus, the data owner can provide UDFs *truncate1* and *truncate2* for the *truncate* operation for columns *IP1* and *IP2* respectively to keep UDFs simple.

As we can see now, data labeling is a straightforward process except two issues, which are discussed below.

4.1 Conditional Labeling

One problem with data labeling is that the proper data categories for some columns cannot be determined statically, but rather depend on what table the current table is joined with. Consider the tables in Fig. 1. The data stored in the *Address* table could be either the customer address or store address since the *Customer* table and the *Store* table both have a foreign key referring to *Address*. Thus, the actual data category for the *Address* table depend on what table it is joined with, which can only be determined when the query is present.

This phenomenon is common for the fact and dimension tables in data warehouse systems. To handle this, we support conditional labeling, i.e., a column is labeled with certain data category based on a *join condition*. A join condition specifies the table is joined with what table on what column(s). For the previous example, the data owner can use conditional labeling to label the *zip* column as follows:

```
LABEL Address.zip WITH Zip WHEN JOIN
      Address.a_id = Customer.c_addr_id
```

This states that the *zip* column of the *Address* table is labeled with the *Zip* category only if this table is joined with the *Customer* table on *a_id=c_addr_id* in a query. In other cases, for example, the *Address* table is accessed without joining with the *Customer* table, this *zip* column will not be labeled with the *Zip* category. The actual data categories for these conditionally labeled columns are determined during the query checking phase, which will be discussed in Section 5.2.

4.2 Complex Data Types

In the previous discussion, we implicitly assume the column is the basic unit for data labeling. However, to handle semi-structured data, distributed data analytics systems such as Spark-SQL also support complex data types including array, struct, and map, which can be nested arbitrarily.

When labeling columns with complex data types, we treat each sub type separately. The basic idea is as follows:

- array: array items can be labeled separately based on array indices.
- struct: for struct, each field is labeled separately since fields usually have different semantics.
- map: for map, currently we support labeling values for predefined keys.

Note that labeling on these columns should also be recursive since the types can be arbitrarily nested.

As a concrete example, consider the following column *c_customer* with struct type which stores customer information:

```
c_customer: struct{
  c_name: string,
  c_gender: string,
  c_address: struct{
    c_state: string,
    c_city: string,
    c_zip: string
  },
  c_contact: map[string, string]
}
```

c_customer is a struct column which contains primitive columns (*c_name*, *c_gender*), a nested struct column *c_address* for storing the customer address information, and a map column *c_contact* for storing the contact information with pre-defined keys such as 'phone' and 'fax'. To label column *c_customer*, the user can label each sub column separately. For example, *c_customer.c_name* can be labeled with category *Name*, while *c_customer.c_address.c_state*, *c_customer.c_address.c_city*, and *c_customer.c_address.c_zip*, can be labeled with categories *State*, *City*, and *Zip* respectively. Similarly, *c_customer.c_contact['phone']* can be labeled with category *Phone*.

Interestingly, even some columns with primitive types may have composite semantics. For example, a *birth* column with the *Date* type may contain all information of categories *birth_year*, *birth_month*, and *birth_day*, which further can be extracted with UDFs *getYear*, *getMonth*, and *getDay* respectively. To enable fine-grained control on composite columns, we treat them similarly as complex types. Specifically, the data owner can label each *extract operation* (UDFs for extracting information) on these columns separately. For example, the extract operation *getYear* on the *birth* column can be labeled as the *birth_year* category, while *getMonth* on the *birth* column as the *birth_month* category.

At last, we discuss a bit more on the difference between desensitize operations and extract operations. The incentives of both operations are quite similar, i.e., to obtain partial information from the input. However, the key difference is that desensitization is irreversible, while it is not the case for extraction. For example, one can easily restore full birth information by calling *getYear*, *getMonth*, and *getDay* on the *birth* column in a same query. Thus, these operations should be treated as extract operations rather than desensitize operations.

5 Policy Enforcement

With the PSpec policy and the labeled data set, we now discuss the policy enforcement process against data analytics queries written in SQL-like languages. Most existing distributed data analytics systems share a similar workflow of query execution. The submitted query is first parsed and analyzed in the master node, e.g., the Driver in Spark-SQL, which is then compiled into a distributed job. Finally, a set of worker nodes concurrently executes the compiled distributed job on partitioned data to transform data and produce final results.

Since we check queries statically, the queries are checked by our policy checker before being compiled into distributed jobs in the master node. The basic idea is to analyze how each data category flows (through what desensitize operation) in a query. To achieve this, we first construct attribute lineage trees to represent the attribute flow, which is then used to extract the data category flow. Finally, the data category flow is checked against the PSpec policy according to the semantics in Section 3.2. Here we assume readers are familiar with query processing in relational database systems, and the reference material can be found in [19]. For ease of discussion, we assume queries have been parsed and optimized as optimized query plans by the query processor.

5.1 Lineage Tree Construction

In the following, we use the term *stored attributes* to refer to the attributes that originate from tables, i.e., columns. Since data categories and desensitize operations are essentially stored attributes and UDFs respectively, we first construct *attribute lineage trees* to track how each stored attribute flows in a query. The leaf nodes of a lineage tree must be stored attributes, while the non-leaf nodes are transformations applied to the children.

Consider the following query that tries to find out the average sales price for female customers in each state and city:

```
SELECT concat(a_state, a_city), AVG(ss_price)
FROM Store_Sales JOIN Customer ON ss_customer_id = c_id
                 JOIN Address  ON c_addr_id = a_id
WHERE c_gender = 'F'
GROUP BY a_state, a_city
```

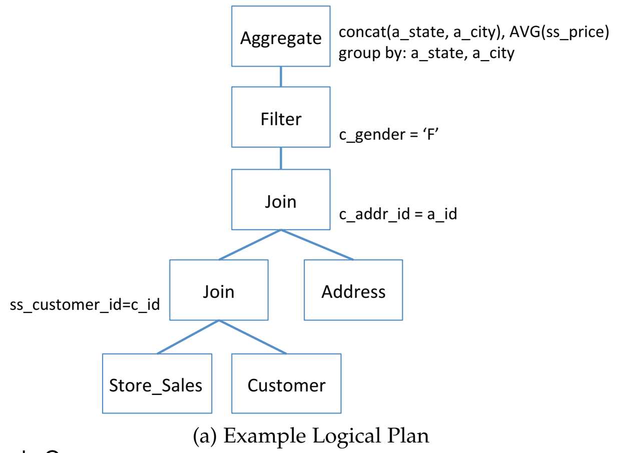
Fig. 6a shows the logical plan for this query generated by Spark-SQL. From the logical plan, we construct the corresponding lineage trees for attributes used in projection and condition as shown in Fig. 6b and Fig. 6c respectively. Since stored attributes are directly used in conditions, the lineage trees in Fig. 6c only contain leaf nodes. With lineage trees, we can clearly see how stored attributes are used in the query. For example, *ss_price* is projected after the AVG function.

Now we discuss how to construct attribute lineage trees from a query plan. Recall that in our system the source data is managed by data owners while analysts are only allowed to access the data for analysis, thus we only consider select queries here. The basic algorithm framework is shown in Fig. 7. Function PROPAGATE takes as input a query plan, and returns a pair of sets which contain the lineage trees for attributes used in projection and condition respectively.

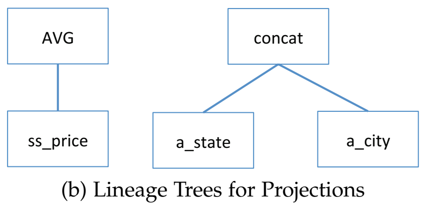
Before the algorithm starts, we attach each plan operator with a map $projections$ that maps each attribute outputted by the plan operator to a lineage tree, and use a global set $conditions$ to track the lineage trees for all attributes used in conditions. Function PROPAGATE performs post-order traversal over the query plan. For each plan operator, it calls function TRANSFORM and function COLLECT to update $plan.projections$ and $conditions$ respectively. Briefly, TRANSFORM calculates map $projections$ based on the attribute transformations in current operator, and COLLECT returns the lineage trees for the attributes used in the predicates in current operator.

The remaining problem is to define functions TRANSFORM and COLLECT for each plan operator. By default, TRANSFORM simply returns map $projections$ of the child operator if the current operator does not perform any attribute transformation, while COLLECT simply returns $\emptyset$ for operators that do not contain any predicate. Example operators include *Limit* and *Distinct* etc. While TRANSFORM and COLLECT for the following operators need special care and will be explained briefly.

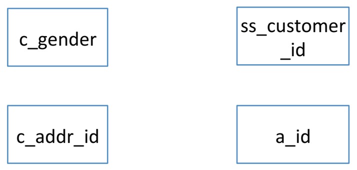
- *Table*: TRANSFORM returns a map that maps each stored attribute to a single node lineage tree whose root is the stored attribute. COLLECT works as default.
- *Project*: TRANSFORM returns a map that maps each attribute defined by the projection list to a lineage tree with the corresponding transformations added on top of the lineage trees for the attributes used to define the new attribute. COLLECT returns the lineage trees for the attributes used in the predicates in the *If* and *CaseWhen* statements (if any). Both statements are supported by SQL. Especially, an *If* statement is of the form: `if` c `then` v_1 `else` v_2, where c is a predicate, v_1 and v_2 are two values; a *CaseWhen* statement is more general and needs to evaluate a set of predicates. When resolving predicates, COLLECT should also add the corresponding transformations on top of the existing lineage trees.
- *Filter*: TRANSFORM works as default. COLLECT returns the lineage trees by resolving the predicates in the filter condition.
- *Join*: TRANSFORM returns the union of map $projections$ of the left and right operators. COLLECT returns the lineage trees by resolving the predicates in the join condition.
- *Aggregate*: *Aggregate* is processed similarly as *Project*, and aggregate functions like COUNT, SUM, AVG, MIN and MAX are simply treated as transformations.
- *Sort*: TRANSFORM works as default. COLLECT returns the lineage trees for the attributes used in the sort expression.
- Binary operators: For binary operators *Union/Intersect/Except*, the output attributes actually combine the corresponding attributes in the left and right operators. Thus, TRANSFORM returns a map that maps each attribute to a lineage tree that combines the lineage trees for the corresponding attributes in



(a) Example Logical Plan



(b) Lineage Trees for Projections



(c) Lineage Trees for Conditions

Fig. 6. Example Query

```
1: $conditions \leftarrow \emptyset$
2: function PROPAGATE($plan$)
3:     for $child$ in $plan.children$ do
4:         PROPAGATE($child$)
5:     $plan.projections \leftarrow$ TRANSFORM($plan$)
6:     $conditions \leftarrow conditions \cup$ COLLECT($plan$)
7:     return $(plan.projections.values, conditions)$
```

Fig. 7. Algorithm for Computing Lineage Trees

the left and right operators with a dummy transformation. COLLECT works as default.

For example, consider the logical plan in Fig. 6a. For table operators *Store_Sales*, *Customer*, and *Address*, $conditions$ is always $\emptyset$, while $projections$ of each operator is initialized as discussed above. For *Join* on *Store_Sales* and *Customer*, $projections$ is the union of $projections$ of *Store_Sales* and *Customer*, while the lineage trees for attributes *ss_customer_id* and *c_id* are added into $conditions$. Another *Join* is processed similarly. Then for *Filter*, $projections$ is inherited from its child, while the lineage tree for attribute *c_gender* is added into $conditions$. Finally for *Aggregate*, the aggregate expression defines two attributes, which are *concat* on *a_state* and *a_city*, and *AVG* on *ss_price*. Thus, $projections$ maps the defined attributes to the lineage trees in Fig. 6b, and $conditions$ contains the lineage trees in Fig. 6c.

Note that for the *CaseWhen* and *If* statements in the *Project* and *Aggregate* operators, the lineage trees for the attributes used in the predicates should be added into $conditions$ only if the defined attributes are projected or used in the subsequent operators. However, for simplicity, we assume the logical plan has been optimized and unused attributes have been removed. Otherwise, one needs to track $conditions$ for each attribute separately as for $projections$.

5.2 Flow Extraction

Since lineage trees are only intermediate structures representing attribute flows, we should then extract data category flows for policy checking. To achieve this, we need to determine the data category for each stored attribute and map UDFs to corresponding desensitize operations. It is straightforward to determine the data categories for statically labeled attributes, while the difficulty here is to determine the data categories for attributes with conditional labels and complex data types as discussed in Section 4.

Resolve Conditional Labels. To resolve the data categories for conditionally labeled attributes, we need to determine which join condition is satisfied by the query. However, we cannot simply check the predicates in the *Join* operator, since the query may filter join results in the *Filter* operator or even the *CaseWhen* and *If* statements. To handle this, we track all equality predicates in the query and build an undirected equality graph with nodes being stored attributes or constants and edges indicating two nodes are equal. Then a join condition is satisfied by a query if the join columns are reachable from one to another in the equality graph.

When tracking the equality predicates in a query, two issues need special care here. First, since a table may be referenced multiple times in a query, we should differentiate the stored attributes from multiple references of a same table. Second, when resolving the equality predicates, approximation is necessary to ensure safety. The principle is that for any equality predicate, we consider the stored attributes or constants appearing in the left and right part equal pair-wisely. Consider predicate $a + b = c$, we need to safely assume both $a = c$ and $b = c$ hold since otherwise the analyst may let b evaluate to 0 at runtime to bypass the mechanism. However, the approximation rarely causes false positives for normal queries.

We further elaborate the process with an example. Consider the following query that finds addresses for both male customers and stores:

```
(SELECT a_state, a_city, a_street
 FROM Address JOIN Customer ON a_id = c_addr_id
 WHERE c_gender = 'M')
INTERSECT
(SELECT a_state, a_city, a_street
 FROM Address JOIN Store
 WHERE a_id = s_addr_id)
```

Suppose the columns in the *Address* table are labeled with the *State*, *City*, and *Street* categories only when the *Address* table is joined with the *Customer* table on *a_id* = *c_addr_id*. We assume stored attributes are equipped with

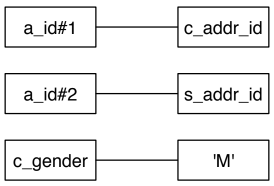


Fig. 8. Example Equality Graph

unique ids based on table references. Thus, the stored attributes from the *Address* table are differentiated by $\{a\_id\#1, a\_state\#1, a\_city\#1, a\_street\#1\}$ and $\{a\_id\#2, a\_state\#2, a\_city\#2, a\_street\#2\}$ (#N indicates attribute id). Fig. 8 then shows the equality graph after analyzing the equality predicates in the query. Finally, after performing reachability analysis on the equality graph, we realize that the join condition for *a_state*#1, *a_city*#1, and *a_street*#1 is satisfied since *a_id#1* = *c_addr_id*, but is not satisfied for *a_state*#2, *a_city*#2, and *a_street*#2.

Resolve Complex Data Types. The basic principle for resolving the data categories for attributes with complex data types is that if a type is directly accessed with no subtype selector, then the data categories for all its subtypes are considered accessed. Consider the example struct column *c_customer* in Section 4.2. If a query accesses *c_customer.c_name*, only category *Name* is considered accessed. However, if a query accesses *c_customer.c_address*, then all categories corresponding to this column, i.e., *State*, *City* and *Address* are considered accessed. Similarly, if column *c_customer* is directly accessed, then all categories corresponding to the subtypes of this column are considered accessed. Attributes with composite semantics are treated similarly, i.e., we identify extract operations performed on these attributes and effective data categories are determined based on identified extract operations. To ensure safety, we consider an extract operation is effective only if it is the direct parent of a stored attribute in the lineage tree.

Resolve Desensitize Operations. After the data categories are determined for stored attributes, we should further identify the desensitize operation performed on each data category. To do so, we first extract the transformation path for each data category by traversing the lineage trees from leaf to root, and then identify which desensitize operation exists in the path. Only the first desensitize operation is effective since we assume a data category can only be desensitized once. For example, consider the lineage trees in Fig. 6. Since *AVG* is a desensitize operation for *Sale_Price*, we know that *Sale_Price* is projected after the *AVG* operation. However, *State* and *City* are directly projected since *concat* is not a desensitize operation for them.

5.3 Query Checking

Recall from Section 3.2 that a query q is modeled as a triple $q = (q.user, q.asso, q.des)$, where $q.user$ is the user category that submits q, $q.asso$ is the associated data access, and $q.des$ is the desensitize operations. In our system, $q.user$ is obtained by assigning each data analyst with a proper user category by the system administrator. After flow extraction, we have already known how each data category d flows to action channel a through what desensitize operation op, which exactly constitutes $q.asso$ and $q.des$.

With $(q.user, q.asso, q.des)$, we can check whether the query satisfies the policy according to the semantics discussed in Section 3.2. If any violation is detected, the query is stopped and the violated rule is returned for the data analyst to revise the query. Otherwise, the query is proceeded to subsequent executions. Thus, our system guarantees only queries satisfying the PSpec policy are executed.

One advantage of our query checking algorithm is that it is totally based on information flow analysis on the query plan without touching the underlying data being queried. We discuss the complexity of our policy checking algorithms in the following. The complexities of lineage tree construction and flow extraction are omitted here, since they are both straightforward one-pass algorithms. While for query checking, in the worst case, it may take $O(d * n^d)$ time to check the rule satisfaction for a query (Definition 3.5), where n is the number of data categories in the vocabulary, and d the length of associated data accesses in the rule. In practice, the query checking algorithm is quite efficient, since the rules rarely refer to long associated data accesses, which makes d a small constant. From our evaluation, checking complex queries can always be finished within milliseconds, which shows it is suitable for big data systems since it only incurs negligible overhead over query execution. Our algorithm do not handle multiple correlated queries, since in practice most queries are independent and linking results from multiple queries together will further exponentially increase the complexity of our algorithm.

6 Implementation and Evaluation

We have implemented PSpec-SQL on top of Spark-SQL 1.5.0 with support for Hive tables [2]. The PSpec parser and label manager are both implemented as standalone modules, while the policy checker is implemented inside the query processing component of Spark-SQL, i.e., the spark-catalyst module. For PSpec, we also developed a graphical authoring tool to facilitate data owners write the PSpec policy. The source code of our prototype implementation is available on GitHub [3]. Note that the policy checker can also be implemented as a separate proxy over existing database systems. The main change is to construct the lineage trees from SQL parse trees instead of query plans.

To evaluate our prototype implementation, we carried out a case study on several industry standard benchmarks obtained from the Transaction Processing Performance Council (TPC) [4], with the focus on usability, performance, and practicability. Each benchmark consists of several database tables and a number of SQL-based queries. All benchmarks contain some customers' privacy information. In the case study, we focus on protecting these customers' information from being improperly used via our PSpec-SQL system.

In practice, a data owner may possess several databases, and he may want to formulate a common privacy policy over all his possessed databases. To reflect this requirement,

2. http://hive.apache.org/
3. https://github.com/thufv/privacy
4. http://www.tpc.org/

we conducted the case study in the following steps. First, we sketched a vocabulary among all benchmarks, and drafted a common PSpec policy. Second, we labeled the relevant columns of tables in these benchmarks with the proper data categories. Finally, we checked the compliance of all queries included in these benchmarks with our PSpec policy. The violated queries were detected and reported by our PSpec-SQL system. The detailed process is discussed below following a description of TPC benchmarks used in our evaluation.

6.1 TPC Benchmarks

The Transaction Processing Performance Council (TPC) is a worldwide consortium whose goal is to define and disseminate industry standard benchmarks for databases and transactional systems [20]. Many top hardware and software vendors, including Microsoft, Intel, IBM, Dell, Hewlett Packard (HP), Cisco etc, are members of TPC. The industry members participate in the development of TPC benchmarks.

There are currently 11 industry benchmarks actively maintained by TPC. Many of these benchmarks are designed to measure performance (like speed, reliability etc) of transaction processing. Since our focus is the privacy protection of data analytics, we require the benchmark contain SQL analytics queries on some customers' information. As a result, three TPC benchmarks are selected, namely TPC-DS [15], TPC-H [20], and TPCx-BB [21].

TPC-DS benchmark [15] is a general benchmark for decision support systems. TPC-DS models a national-wide retail company, which sells goods through stores, catalogs, and the Internet. It includes a database schema (7 fact tables and 17 dimension tables), which is organized as a snowflake schema, and 99 queries. For customer-related information, TPC-DS stores the customer data into the following tables: *Customer*, *Customer_Address*, *Customer_Demographics*. It also stores the items bought by customers through *Sales* tables.

TPC-H benchmark [20] is another benchmark for decision support systems that allow concurrent data modifications. It models a wholesale supplier that sells or distributes products worldwide. TPC-H contains 8 base tables and 21 complex SQL queries. It stores the customers' data into the *Customer* table, and the customers' orders into the *Orders* table.

Finally, TPCx-BB benchmark [21] is designed for benchmarking big data analytics systems such as Spark-SQL [3] and Hive [4]. It models a similar national-wide retail company to that of TPC-DS benchmark. However, it emphasizes the usage of User-Defined Functions (UDFs) to perform more complex data analysis tasks. It further stores the customer's click streams into the *Web_Clickstreams* and *Web_Page* tables to keep track of the online activities of customers. For the query-side, TPCx-BB contains 30 queries with the emphasis of complex data analysis and machine learning tasks.

6.2 PSpec Policy

Note that some tables of these benchmarks store and use customers' personal information. To ensure these privacy-related data are properly used, we made a sample PSpec policy and the process is illustrated as follows.

TABLE 2
Supported Desensitize Operations

Data Category	Supported Desensitize Operations
All	count
Zip	truncate
Phone	substr
URL	substr
Income	range
Vehicle	isZero
Price	sum, avg, min, max

6.2.1 *Vocabulary*

For simplicity, we assume there is only one user category, named *Analyst*. Based on the customer data stored in the three benchmarks, we sketched a hierarchy of data categories, shown in Fig. 9. Most data categories in Fig. 9 are self-explanatory, and abbreviations are used when their meanings are clear. Similar to previous work [2], [22], the customer-related information is classified as *KA* (Key Attribute), *QI* (Quasi Identifier), and *SA* (Sensitive Attribute), which are further divided into finer-grained categories.

The supported desensitize operations for some data categories are shown in Table 2. Note that we have not identified too many desensitize operations since the schemes of these benchmarks are well designed and most of their columns already have fine-grained meanings. Thus, the PSpec rules can directly refer these fine-grained categories without desensitize operations. In fact, desensitize operations will be more required when columns have coarse-grained semantics.

6.2.2 *PSpec Rules*

After identifying user and data categories, we wrote some PSpec rules to ensure that the privacy-related data is properly used. These rules are listed in Table 3, where each rule specifies a data usage restriction to the customers' information.

The first three rules forbid the direct access of KAs, since KAs can directly locate individuals. Among the three KAs, the *ID* and *Email* attributes are completely forbidden to be accessed, and the *Phone* attribute requires a desensitize operation. Among the many QIs, the *Street* and *Name* attributes can also be used to identify an individual, and are thus forbidden to be accessed (in rules 4 to 5). Rules 6 to 8 forbid the associated access of several combinations of QIs since each of these combinations has also a great chance to locate individuals. Rule 9 requires a desensitize operation before the access of *Zip*. In [2], a sophisticated scene was discussed, where linking QIs with some public datasets can also locate individuals. Rules 10 to 13 were used to avoid such kind of privacy breach. Finally, rules 14 to 18 regulate that some sensitive attributes must be well desensitized, when they are accessed in combination with any quasi identifier.

The above PSpec rules were mainly specified by a junior undergraduate student (who have received *one-hour* training in PSpec, and is familiar with the TPC benchmarks). This student succeeded in writing most of these PSpec rules in *three-hours*. This illustrates the usability of the PSpec language.

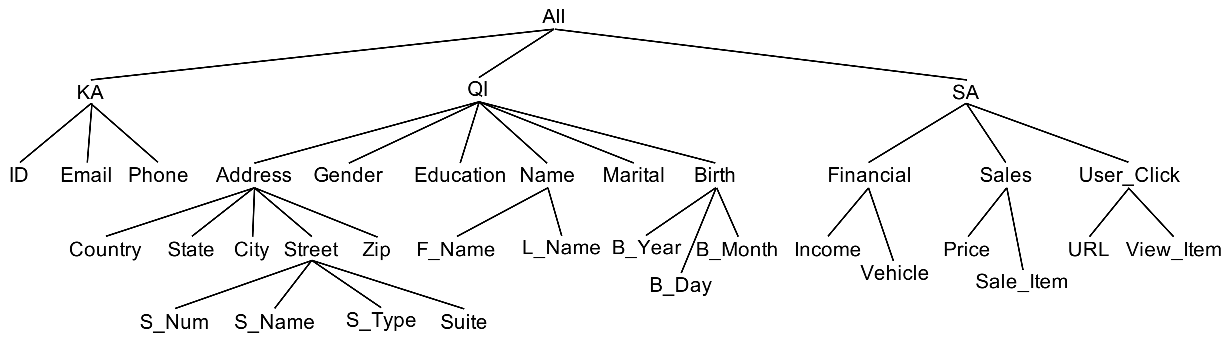


Fig. 9. Data Hierarchy Used in Case Study

TABLE 3
PSpec rules for TPC benchmarks

ID	Rule
1	Analyst,[access ID]=>forbid
2	Analyst,[access Email]=>forbid
3	Analyst,[access Phone]=>[substr]
4	Analyst,[projection Street]=>forbid
5	Analyst,[projection Name]=>forbid
6	Analyst,[access S_Num,access S_Name, access Suite]=>forbid
7	Analyst,[access F_Name,access L_Name]=>forbid
8	Analyst,[access B_Day,access B_Month, access B_Year]=>forbid
9	Analyst,[access Zip]=>[{truncate}]
10	Analyst,[projection Birth,projection Address, projection Gender]=>forbid
11	Analyst,[projection Birth,projection Address, projection Marital]=>forbid
12	Analyst,[projection Birth,projection Address, projection Education]=>forbid
13	Analyst,[projection Birth,projection Address, projection Financial]=>forbid
14	Analyst,[access QI,access Vehicle]=>[{},{isZero}]
15	Analyst,[access QI,access Income]=>[{},{range}]
16	Analyst,[access QI,access Price] =>[{},{sum,count,avg,min,max}]
17	Analyst,[access QI, projection View_Item] =>[{},{count}]
18	Analyst,[access QI, projection URL] =>[{},{count, substr}]

It should also be noted that Table 3 may not be a complete set of rules for protecting privacy of the benchmarks. Actually, this case study is not for privacy specification, but rather for evaluating the PSpec language. User privacy may subject to different regulations in different organizations and scenarios. In practice, the data owner should adjust the PSpec rules to protect user privacy based on the actual policy requirements and regulations.

6.2.3 Data Labeling

With the data categories, we then labeled the table columns in these three benchmarks with proper data categories. We also implemented desensitize operations, including *truncate*, *range*, and *isZero*, as UDFs and registered them into our system.

In general, the data labeling will not incur too much effort. Many tables of these benchmarks can be skipped in this process, since they are irrelevant to the customers' information. Even for tables that contain customers' information, many of their columns are with primitive types (for example, the *Customer* table of TPC-DS), and thus can be directly labeled based on their semantics.

However, due to our case study spanning several databases, there are several complex scenes that need more careful handling. First, the same information may be stored at different granularity in different databases. For example, TPC-H stores the customer address using a single column named *Address*, while TPC-DS divides the customer address into several parts and stores them in *State*, *City* and *Street* columns, respectively. To address this, we simply labeled the *Address* column of TPC-H with a tuple (*State*, *City*, *Stree*) of data categories, reflecting the composite semantics of this column. Second, the information stored in a column maybe multi-fold. For example, the *Name* column of the *Nation* table in TPC-H stores the nationality information for both suppliers and customers, but we may only concern the customers' nationalities. To resolve this, we used conditional labeling to label this column only when the *Nation* table is joined with the *Customer* table. Another example is the *i_current_price* column of the *Item* table in TPC-DS. Its columns should be labeled with the *Price* and *Item* categories only when it is joined with the *sale/return* tables, since we only care about items that customers have bought or returned.

6.3 Query Checking

There are in total 150 queries among all three benchmarks. These queries range from reporting queries, ad hoc queries, iterative queries to data mining queries. For example, TPC-DS contains 99 queries, each of which corresponds to a particular business question. As we have counted, the simplest query contains 18 lines of code (formatted) and refers only 1 table, while the most complex query contains 456 lines of code and refers 14 tables.

Some queries contain some syntax that is not supported by Spark-SQL, such as window functions, the WITH statement and sub queries in the WHERE statement, we transform these queries into their equivalences that can be processed by Spark-SQL (the window function is an exception, which was simply removed). The transformed results can be found in our GitHub repository. Moreover, some queries in the TPCx-BB benchmarks integrate Python codes to perform machine learning-related analysis. These Python codes were simply treated as uninterpreted UDFs.

All queries were submitted to our PSpec-SQL system to check their compliance with the sample PSpec policy. We measured the time cost of policy checking on each query. All queries were efficiently checked. The shortest takes only *1ms* and the longest takes *96ms* with the average time being *8ms*. Our query checking algorithm is totally based on the information flow analysis without touching the underlying data, it only incurs negligible overhead and is thus suitable for online data analysis.

Finally, our PSpec-SQL system reported *25 policy violations*, among which 21 were caused by queries of TPC-DS. Note that TPC-DS is a comprehensive benchmark with various kinds of data analytics queries. Its queries 23, 24, 30, 34, 38, 46, 68, 73, 74, 81, and 84 attempt to directly project the customer's first name and last name, thus violate rules 5 and 7 in Table 3. Moreover, its queries 30 and 81 even attempt to project the customer's birth day, email, or living street in combination with the customer name, which can cause severe privacy violations. Other violations, including queries 13, 15, 45, 48, 64, and 85 of TPC-DS, directly access *Price*, *Income*, or *Zip* without desensitization as required.

For queries of TPC-H and TPCx-BB, we found only 4 violations. The reason is that queries in these two benchmarks mainly focus on analyzing the general trend of sales and products, without accessing the customer information in detail. But still, queries 10 and 18 of TPC-H, queries 6 and 13 of TPCx-BB directly output customer names, and thus violate rules 5 and 7 in Table 3. Moreover, the query 10 of TPC-H also attempts to output the customer's address and phone information, thus violates rules 3 and 4, too. The query 6 of TPCx-BB also attempts to output the customer's country and email information, thus violates rules 2 and 4, too.

Among the 25 reported violations, 4 are false positives. These false positives all belong to TPC-DS and are all caused by unsupported desensitize operations. For example, queries 8 and 19 of TPC-DS desensitize *Zip* with *substr* rather than the specified *truncate* operation. Since *substr* is not a supported desensitize operation of *Zip* (see Table 2), these queries were recognized as violations by our system. Similarly, queries 34 and 74 of TPC-DS directly check whether *Vehicle* is greater than zero in their conditions, but not using the specified *isZero* operation. All these false positives can be easily eliminated by simply revising the queries to use the appropriate UDFs.

7 Discussion

In this section, we briefly discuss some possible extensions of our system.

PSpec Elements. *Purpose*, *condition*, and *obligation* are important concepts in privacy policies [23]. In PSpec, data desensitization can be viewed as a special *condition* that must be satisfied by queries in order to access data. For ease of discussion and policy enforcement, we have intended to leave out the *purpose* and *obligation* elements. However, it is straightforward to extend the syntax and semantics of PSpec with purposes and obligations to express data can only be accessed for certain purposes or what should be performed after a query is executed.

Record-level Control. As records of a same table may have different requirements, an interesting extension of our system is to support record-level control. Here we briefly discuss the basic idea, while postpone its implementation as a future work. Following the idea of predicated grants [24], we can extend the PSpec rules with an extra *predicate* element, which defines a predicate over data categories to specify accessible records. An example predicate could be "only customers with ages greater than 18 can be accessed". During the privacy checking phase, we first need to transform the defined predicates into concrete predicates over database columns, which can then be easily enforced by rewriting the filter condition of a query to filter out inaccessible records.

Automatic Data Labeling. To simplify users' effort, another possible extension of our system is to automatically label the relational data based on their semantics. By treating the data hierarchy as a special schema, many previous works on schema mapping [25], [26] can be referred for implementing this automatic labeling approach. To improve precision, it is also worth considering making this approach semi-automatic, i.e., let the user guide the labeling process but with minimal extra effort.

8 Related Works

In this section, we discuss several related works, which fall into the following categories.

Information Flow Analysis. Information flow analysis on programs is originated from the security community to check whether programs leak sensitive information [17], [18]. This technique has been widely used in the security community, ranging from certificating secure programs [17], preventing security attacks [27], and detecting privacy leakages in mobile applications [28]. Compared with these works, we use information flow analysis for a different purpose, i.e., checking privacy compliance of SQL queries.

Access Control Language. In the past decades, many policy specification languages have been proposed to formalize text-based privacy and security policies. P3P [29] is a privacy language for formalizing website privacy policies with the focus on how personal information is collected, shared and retained. EPAL [7] targets at formalizing and enforcing privacy policies within enterprises. XACML [8] is an OASIS standard access control language, which supports user-defined attributes and various policy/rule combining algorithms for different scenarios. Some of the previous works have motivated the design of PSpec, but PSpec differs from them in several aspects. First, PSpec is designed as a specific privacy language suitable for data analytics systems. Second, most of the previous works consider data elements independent of each other, which does not hold in data analytics. PSpec provides explicitly support for associated data access to handle this.

Database Access Control. Another category of the related works is database access control, as our work naturally applies to traditional database systems. The sequence of works [9], [30], [10] proposed the idea of Hippocratic database. Query auditing techniques [31], [32] audit submitted queries to detect whether sensitive information is leaked. The works [11], [12] enforce purpose-based policies within database systems. Colombo and Ferrari [13] further consider action-aware policies, i.e., actions performed by queries on certain categories of data. DataLawyaer [33] enforces general data use policies encoded into SQL-like queries. Recent works [34], [35] further enforce purpose-based policy within NoSQL data stores, e.g., MongoDB. However, we differ from these works in that we provide a high-level specification language to hide the details of data models, while these works directly operate on the concrete data models, e.g., relational data or NoSQL data. Lacking the high-level specification language may cause barriers for legal experts, who often have no background in database systems.

Privacy-Preserving Systems. The last line of the related works falls into privacy-preserving systems. PINQ [36] is a privacy-preserving data analysis platform built on top of LINQ and differential privacy. Airavat [37] extends MapReduce [38] and integrates mandatory access control and differential privacy to provide security and privacy guarantees. GUPT [39] enforces differential privacy for arbitrary computations with the sample and aggregate framework [40]. The major difference is that our system aims at enforcing privacy policies by specifying data usage restrictions, while these systems mainly rely on differential privacy. As we have already discussed in Section 1, data usage specification is (at least currently) more practical than differential privacy for business data sharing.

Recently, Sen et al. [14] presented a system for auditing privacy compliance for big data systems within organizations. The proposed system is composed of two modules: LEGALEASE for specifying privacy policies and GROK serving as a data inventory. Our work differs from [14] in that we mainly target at checking SQL queries in an online manner and provide comprehensive support for relational data. The system in [14] is designed for handling general privacy-related data use policies, i.e., store, use, and share, and can only operate offline.

9 CONCLUSION

In this paper, we present PSpec-SQL, a distributed data analytics system that automatically enforces privacy compliance for SQL queries. Our system offers a high-level specification language PSpec for data owners to specify their data usage policies, and provides comprehensive support for labeling relational data. Finally, our system checks submitted queries at runtime for compliance to ensure only policy-compliant queries are executed.

10 ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61672310 and Grant No. 61527812, the National Science and Technology Major Project under Grant No. 2016ZX01038101.

REFERENCES

[1] H. Kazeli, "Cloud business intelligence," in *Business Information Systems Workshops*. Springer, 2014, pp. 307–317.

[2] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[5] C. Dwork, "Differential privacy," in *Automata, languages and programming*. Springer, 2006, pp. 1–12.

[6] ——, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation*. Springer, 2008, pp. 1–19.

[7] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal 1.2)," *Submission to W3C*, 2003.

[8] T. Moses *et al.*, "Extensible access control markup language (xacml) version 2.0," *Oasis Standard*, vol. 200502, 2005.

[9] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 143–154.

[10] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending relational database systems to automatically enforce privacy policies," in *ICDE*, vol. 5, 2005, pp. 1013–1022.

[11] J.-W. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *The VLDB Journal*, vol. 17, no. 4, pp. 603–619, 2008.

[12] P. Colombo and E. Ferrari, "Enforcement of purpose based access control within relational database management systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 11, pp. 2703–2716, 2014.

[13] ——, "Efficient enforcement of action-aware purpose-based access control within relational database management systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 27, no. 8, pp. 2134–2147, Aug 2015.

[14] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in big data systems," in *Proceedings of the 35th IEEE Symposium on Security & Privacy (Oakland)*, 2014.

[15] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 1049–1058.

[16] C. Luo, F. He, D. Yan, D. Zhang, X. Zhou, and B.-Y. Wang, "Pspec: A formal specification language for fine-grained control on distributed data analytics," in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017, pp. 300–302.

[17] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[18] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. IEEE, 2005, pp. 255–269.

[19] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill New York, 1997, vol. 4.

[20] R. Nambiar, N. Wakou, F. Carman, and M. Majdalany, "Transaction processing performance council (TPC): State of the council 2010," in *Proceedings of the Second TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*, ser. TPCTC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–9.

[21] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: towards an industry standard benchmark for big data analytics," in *ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 1197–1208.

[22] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 3, 2007.

[23] S. Fischer-Hübner, *IT-security and privacy: design and use of privacy-enhancing security mechanisms*. Springer-Verlag, 2001.

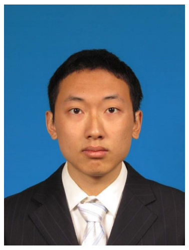
[24] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine grained authorization through predicated grants," in *IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 1174–1183.

[25] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," in *VLDB*, vol. 1, 2001, pp. 49–58.

[26] E. Rahm and P. A. Bernstein, “A survey of approaches to automatic schema matching,” *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.
[27] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: Dynamic taint analysis with targeted control-flow propagation.” in *NDSS*, 2011.
[28] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones,” *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.
[29] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle, “The platform for privacy preferences 1.0 (p3p1. 0) specification,” *W3C recommendation*, vol. 16, 2002.
[30] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, “Limiting disclosure in hippocratic databases,” in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 108–119.
[31] R. Motwani, S. U. Nabar, and D. Thomas, “Auditing sql queries,” in *IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 287–296.
[32] R. Kaushik and R. Ramamurthy, “Efficient auditing for complex sql queries,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 697–708.
[33] P. Upadhyaya, M. Balazinska, and D. Suciu, “Automatic enforcement of data use policies with datalawyer,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 213–225.
[34] P. Colombo and E. Ferrari, “Enhancing mongodb with purpose based access control,” *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2015.
[35] ——, “Fine-grained access control within nosql document-oriented datastores,” *Data Science and Engineering*, vol. 1, no. 3, pp. 127–138, 2016.
[36] F. D. McSherry, “Privacy integrated queries: an extensible platform for privacy-preserving data analysis,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 19–30.
[37] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, “Airavat: Security and privacy for mapreduce.” in *NSDI*, vol. 10, 2010, pp. 297–312.
[38] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[39] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, “Gupt: privacy preserving data analysis made easy,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 349–360.
[40] K. Nissim, S. Raskhodnikova, and A. Smith, “Smooth sensitivity and sampling in private data analysis,” in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. ACM, 2007, pp. 75–84.

Chen Luo received the B.S. degree from Tongji University in 2013, and the M.S. from Tsinghua University in 2016. He is currently a PhD student in University of California Irvine, US. His research interests include formal methods and database systems.

Fei He received the B.S. degree in computer science and technology from National University of Defense Technology in 2002, and the Ph.D. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.



Fei Peng received the B.S. degree from Tsinghua University in 2016. He is currently a Master student in Columbia University. His research interests include formal methods and program verification.

Dong Yan received the Ph.D. degree in computer science and technology from Tsinghua University in 2015. He worked at Intel Labs China as a data analysis platform researcher from 2015 to 2016. He is currently a Postdoc in Tsinghua University. His research interests include distributed system, big data and large-scale machine learning.

Dan Zhang received the Ph.D. degree in computer science and technology from Xi’an Jiaotong University in 2006. He worked at Intel Labs China as research scientist from 2006 to 2016. He is currently Architect in UISEE corporation. His research interests include autonomous driving, big data, data privacy preservation, robotics, compiler, etc.

Xin Zhou received the Master degree in Computer Science from Fudan Univerisy in 2002. He worked at Intel Lab China as a senior researcher until 2016. He is currently the product general manager of UISEE corporation. His research interests include machine learning, big data and privacy.